Steady-state Optimization of Modelica Models and Functional Mockup Units with Pyomo

Jesse Gohl[1] Hubertus Tummescheit[1] Robin Andersson[1]
Matthew Stuber[2]

[1]Modeon Inc, USA,
[2]Dept. of Chemical & Biomolecular Engineering, University of Connecticut
jesse.gohl@modelon.com

Abstract

This paper describes two ways on how to interface Functional Mockup Units (FMUs) and Modelica models through the Pyomo's foreign function interface with Pyomo. Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities. Modelica has arguably much better modeling capabilities than Pyomo, but Pyomo integrates excellent optimization solvers, such as Ipopt (Wächter et al. 2006), and provides a good optimization infrastructure. The Interface has been developed in the context of a NAWI, (National Alliance Water Innovation) Hub project in collaboration with the University of Connecticut and Sandia National Labs. The optimization has been set up and tested within Modelon's Modelica platform Modelon Impact. An unpublished, detailed multi-effect desalination plant developed by Prof. Matt Stuber in the context of (Stuber et al., 2015) has been used to demonstrate the capabilities, as well as simple test models, and design models from Modelon's commercial Libraries.

Keywords: Modelica, Functional Mockup Interface, FMI, Steady-state Optimization, Design Optimization

1 Introduction

There is a growing list of options to perform optimization studies involving Modelica models. A number of simulation tools support optimization natively within their simulation environment (OpenModelica, Ansys Twin Builder®, System Modeler, the Modelica Optimization Library, etc.). The models can also be exported as Functional Mockup Units (FMU) and imported to specialized optimization tools (modeFrontier®, Optimus®, etc.). A couple of dynamic optimization (Bryson 1999) methods (OpenModelica and JModelica) rely on CasADi (Andersson 2011; Bachmann 2012; Ruge 2014). The tools transfer the Modelica model to CasADi for automatic differentiation and optimization. Originally this was done via an XML file format for both tools (Magnusson 2015), but the Optimica Compiler Toolkit (OCT) has evolved from JModelica to support more comprehensive coverage of the Modelica language, transferring large parts of the language into a native CasADi problem (Modelon 2024). This is done automatically but both methods rely on sufficiently restricted models to avoid unsupported constructs by CasADi. The Optimica Compiler Toolkit also includes support for derivative free optimization using the Nelder-Mead simplex method (Nelder and Mead 1965; Fletcher 1987) for static optimization. In addition, OpenModelica and OCT support the Optimica® language for the description of the optimization problem. This language is an extension to the Modelica language. An alternative to the above methods that is explored in this paper, is to connect the FMU to a solver through the Pyomo Python toolbox for optimization and solution through its connection to IPOPT.

The Functional Mockup Interface (FMI, Modelica Association 2024) is a standardized, widely accepted API implemented by more than 200 simulation tools for executable simulation models. However, it has been designed for transient simulations, not steady-state (design-oriented) simulation models. It can be used to compute stationary points for transient system models, but the API lacks functions to compute sensitivities with respect to decision variables in an optimization problem symbolically. It is possible, but less accurate and performant to approximate the derivates by finite differences. Modelon Impact offers to convert Modelica model parameters to inputs when translated into an FMU. This allows using the symbolic derivatives with respect to inputs in the standard FMI-interface in the solution process. While this is helpful, and allows to optimize arbitrary Modelica-models, it is not sufficient to robustly optimize large and highly non-linear problems.

This paper will present two different interfaces between Modelica models and Pyomo's optimization algorithms:

1. An interface based on generic FMUs, enhanced with Modelon Impact's capability to convert parameters to inputs for improved accuracy of Jacobian computations.

2. Modelon Impact's internal interface for solving large non-linear systems for steady-state design problems.

Modelon's interface is similar to FMI but adds the ability to couple a Modelica model to an external steady-state solver. This is in spirit like a model exchange FMU.

The first interface allows to couple FMUs for any FMI-compliant tool to Pyomo, the second interface is more robust, able to solve larger models, but is only available for Modelica models in Modelon Impact.

2 Requirements for efficient gradient-based optimization

A typical constrained static optimization problem can be written as follows:

$$\begin{gathered}\text{Minimize } f(x) \\ \text{Subject to } g_j(x) \leq 0, j = 1, \dots, p \\ h_i(x) = 0, i = 1, \dots, m\end{gathered} \tag{1}$$

where $g_j, h_i\colon \mathrm{R}^n \to \mathrm{R}$ are inequality and equality constraints respectively, and $f(x)$ is the objective function. In the context of models coming from Modelica, usually all equality constraints come from the model equations, and the objective function and inequality constraints must be handled outside of the standard Modelica language. For this prototype interface, we have chosen to not use the Optimica Modelica extension proposed by (Åkesson et. al., 2011). A graphical Modelica user environment such as Modelon Impact can allow those to be entered in the definition of the optimization experiment.

With the optimization of a model in Modelica, there are two fundamentally different ways of solving a steady-state optimization problem: 1) the nonlinear solver is the same that is used for steady-state initialization in a Modelica simulator, and the optimization solver is "wrapped" around that, i.e. nested solvers, or 2) the optimization solver handles everything, and treats the model equations as equality constraints ($h_i(x)$), i.e. a single solver. With generic FMUs, a nested solver is the only option. With Modelon Impact's steady-state interface, the second option is possible, and has demonstrably been the faster and more robust option in our testing.

It should be noted that the dimension m of $h_i(x)$ can be large, several thousand equations, which can lead to excessive solution times. However, so called tearing of equation systems (Baharev et al., 2016) can dramatically reduce the dimension, is generally used by Modelica tools, and is also used by us in this interface to reduce the dimension of the $\boldsymbol{h}(\boldsymbol{x})$ vector equations that are exposed to the solver. Also note that, even after the dimensionality reduction through tearing, the resulting subset of the equations is still large enough to justify the use of sparse solver interfaces.

There are further important numerical requirements for efficiency and robustness, some of which are on the model-side of the interface, and some on the optimization solver side of the interface. Pyomo offers methods to compute first and second derivatives for constraints and objective function. On the model side, these can be approximated numerically, or computed analytically. For standard, transient FMUs, they must be approximated numerically. Modelon's dedicated steady-state, FMU-like interface allows to provide even analytic Jacobians to the Pyomo external function interface. Scaling of variables is also important for robustness and can be achieved by making use of the Modelica and FMI feature of nominal values for variables. Note that scaling is much more important than in the simulation case.

3 The Pyomo/PyNumero External Function Interface

The core of both interfaces of this work, mentioned in the introduction, extends from the `ExternalGreyBoxModel` class of PyNumero (Rodriguez et al. 2024). This class allows users to use external models with Pyomo. The class translates the external model, usually written in Python code, into a Pyomo model by wrapping a set of standard methods, necessary to define an optimization problem. Examples of these methods are `evaluate_outputs`, `evaluate_equality_constraints`, `evaluate_jacobian_outputs`, and `evaluate_jacobian_equality_constraints`. The second derivatives can also be defined using the `evaluate_hessian_outputs` and `evaluate_hessian_equality_constraints` methods. The new classes from this effort, wrap (evaluate) the FMI interface methods from a PyFMI FMU object within these PyNumero methods.

The main class constructor of this work accepts a PyFMI FMU object or a steady state FMUProblem[1] object from Modelon's steady-state interface class. The methods provided by these objects are evaluated within the PyNumero methods during the solution of the optimization or static problem. The constructor also accepts lists of relevant variables for inputs, outputs, and constraints, as well as modifiers to FMU parameters and some additional options specific to this interface. The input, output, and constraint lists identify the relevant

[1] The steady state interface in PyFMI follows to the largest extent possible the FMI-Standard 2.0, but it defines a steady-state problem, but a transient one as "normal" FMUs.

FMU variables for the optimization or steady state problem.

4 The Interfaces

In general, FMUs define a mathematical representation of an engineering problem. Frequently, this is defined as an experiment that simulates a transient, progression in time, as a differential-algebraic equation (DAE) system. The *GenericFMU* class from the new Python package simulates an FMU from the initial time to the final time to supply the values needed by an optimization solver. This includes the outputs, constraint residuals, and their derivatives. Because many FMUs support these methods, this allows the new interface to support the widest range of FMUs, regardless of the generation tool.

Static problems do not include a dependence on time because time and derivatives with respect to time do not appear in the DAE system. This means that the mathematical representation of the problem is reduced to a set of nonlinear equations with one or more solutions (hopefully!) that needs to be solved by a nonlinear programming (NLP) solver. Modelon's steady state problem interface is an efficient translation of these types of problems to pass to an NLP solver. Evaluating the outputs and constraints does not require stepping forward in time and is the simplest and most efficient case because, since there are no dynamic states (no time derivatives), the outputs only depend on the inputs in continuous-time mode (as defined by the FMI specification). This means the solver can update the inputs and the outputs can be evaluated without additional method calls. This is also the most efficient case for the evaluation of derivatives and can take advantage of the directional derivatives defined by the FMU. This applies when the inputs to the optimization problem are also the inputs to the FMU (v_{known} of section 2.1.9 of the FMI specification version 2.0.4). The methods of the *StaticFMU* class avoid full simulations to compute the values needed by the optimization solver.

The static interface also supports other cases that include FMUs with event indicators, optimization inputs that are not also FMU inputs (usually this means causality = parameter), and co-simulation FMUs. The static interface checks for these cases and defines the appropriate evaluation methods depending on the situation. For the case of FMUs with event indicators an event handling loop is activated if an event is detected by the indicators. The allowed modes for modifying parameter values are more restricted. For example, normal parameters (e.g. causality = parameter and variability = fixed), can only be modified before exiting initialization mode. Therefore this version of the interface package resets the FMU, sets the updated parameter values, then initializes the FMU to ensure consistency. Co-simulation type FMUs use a similar strategy of setting values and initializing.

5 Test Models and Results

PyNumero includes an example of a problem based on a continuously stirred reactor as described in section 7.4.4 of Bynum et al., (2021). The problem is to find the incoming flow rate of an inlet stream with a single species to a chamber with competing reactions, that maximizes the concentration of a specific species in the outlet stream.

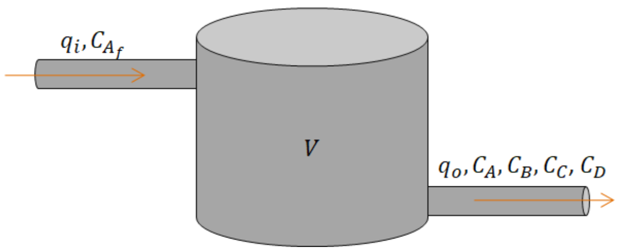


Figure 1. Continuously stirred reactor diagram

The reactions within the vessel are governed by the rates k_1, k_2, and k_3 through the following sequences.

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C \tag{2}$$
$$2A \xrightarrow{k_3} D \tag{3}$$

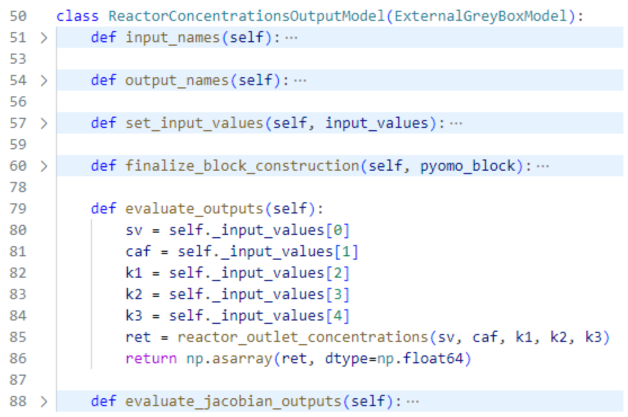
This can be represented by the following model that describes the rates of change of the outlet concentrations C_A, C_B, C_C, and C_D.

$$\dot{C}_A = \frac{q}{V} C_{A_f} - \frac{q}{V} C_A - k_1 C_A - 2k_3 C_A^2 \tag{4}$$
$$\dot{C}_B = -\frac{q}{V} C_B + k_1 C_A - k_2 C_B \tag{5}$$
$$\dot{C}_C = -\frac{q}{V} C_C + k_2 C_B \tag{6}$$
$$\dot{C}_D = -\frac{q}{V} C_D + k_3 C_A^2 \tag{7}$$
$$\frac{q}{V} = sv \tag{8}$$

At steady state, the rates of change of concentrations are zero and the inlet and outlet flow rates are equal. This is equivalent to a statics problem that does not involve time.

$$\dot{C}_X \to 0 \tag{9}$$
$$q_o \to q_i \tag{10}$$

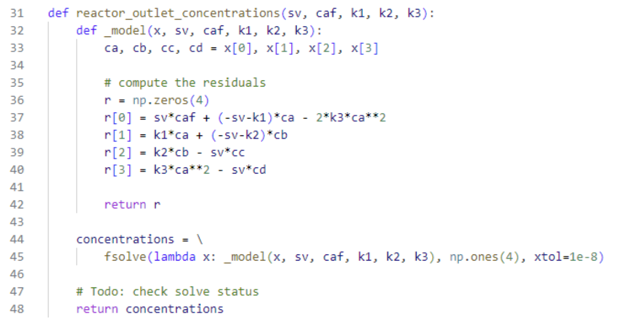
Notice the steady state problem involves a quadratic term for C_A. Depending on the tool, this can require iteration to solve numerically. The example in PyNumero encodes this problem in Python code to demonstrate the external interface to a Pyomo model. The `ExternalGreyBoxModel` of this example appears as in the following image that shows the method definitions from the base class:



```
class ReactorConcentrationsOutputModel(ExternalGreyBoxModel):
    def input_names(self): ...

    def output_names(self): ...

    def set_input_values(self, input_values): ...

    def finalize_block_construction(self, pyomo_block): ...

    def evaluate_outputs(self):
        sv = self._input_values[0]
        caf = self._input_values[1]
        k1 = self._input_values[2]
        k2 = self._input_values[3]
        k3 = self._input_values[4]
        ret = reactor_outlet_concentrations(sv, caf, k1, k2, k3)
        return np.asarray(ret, dtype=np.float64)

    def evaluate_jacobian_outputs(self): ...
```

Figure 2. PyNumero reactor model example Python class definition

The `finalize_block_construction` method is the location for specifying the Pyomo variable attributes like bounds and starting values. Notice the variable bounds are specified separately from the constraints. In this example a lower bound of zero is defined for all the concentrations. Evaluation of the residuals occurs in the `_model` function, defined in the function `reactor_outlet_concentrations`. This is where equations (4) – (8) are defined in the Python model. As the problem is written, solution of this system requires an iterative approach to drive the four residual values toward zero. This relies on *fsolve* from scipy.optimize to minimize the residuals. Whenever the optimization solver needs the values of the outputs (or to estimate the Jacobian as described next), the external NLP solver, *fsolve*, must be called to find the values of `ca`, `cb`, `cc`, and `cd` that result in sufficiently small residual values. This structure can also be used with the FMU based approach if the FMU requires iteration to resolve nonlinear systems of equations. Alternatively, the variables and residual values can be exposed to the optimization solver and the residuals handled as equality constraints (with equality zero). This is possible with the steady-state FMU interface package and is usually considered more efficient and robust.

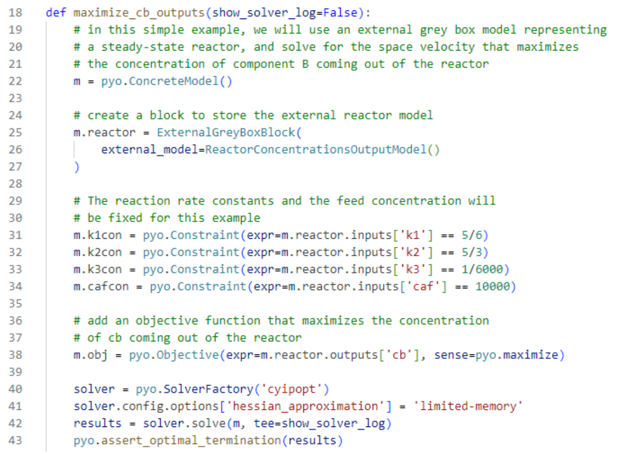


```
def reactor_outlet_concentrations(sv, caf, k1, k2, k3):
    def _model(x, sv, caf, k1, k2, k3):
        ca, cb, cc, cd = x[0], x[1], x[2], x[3]

        # compute the residuals
        r = np.zeros(4)
        r[0] = sv*caf + (-sv-k1)*ca - 2*k3*ca**2
        r[1] = k1*ca + (-sv-k2)*cb
        r[2] = k2*cb - sv*cc
        r[3] = k3*ca**2 - sv*cd

        return r

    concentrations = \
        fsolve(lambda x: _model(x, sv, caf, k1, k2, k3), np.ones(4), xtol=1e-8)

    # Todo: check solve status
    return concentrations
```

Figure 3. PyNumero reactor model output evaluation function

The Jacobian is provided by the `evaluate_jacobian_outputs` method of the reactor class. The matrix is returned in the sparse coordinate format as an instance of a *coo_matrix* class from the scipy.sparse package. The PyNumero example estimates the matrix using finite differences, but it could also have been analytically computed from the model equations.

The Pyomo code to define the optimization problem to maximize the concentration of species B appears as in the following:



```
def maximize_cb_outputs(show_solver_log=False):
    # in this simple example, we will use an external grey box model representing
    # a steady-state reactor, and solve for the space velocity that maximizes
    # the concentration of component B coming out of the reactor
    m = pyo.ConcreteModel()

    # create a block to store the external reactor model
    m.reactor = ExternalGreyBoxBlock(
        external_model=ReactorConcentrationsOutputModel()
    )

    # The reaction rate constants and the feed concentration will
    # be fixed for this example
    m.k1con = pyo.Constraint(expr=m.reactor.inputs['k1'] == 5/6)
    m.k2con = pyo.Constraint(expr=m.reactor.inputs['k2'] == 5/3)
    m.k3con = pyo.Constraint(expr=m.reactor.inputs['k3'] == 1/6000)
    m.cafcon = pyo.Constraint(expr=m.reactor.inputs['caf'] == 10000)

    # add an objective function that maximizes the concentration
    # of cb coming out of the reactor
    m.obj = pyo.Objective(expr=m.reactor.outputs['cb'], sense=pyo.maximize)

    solver = pyo.SolverFactory('cyipopt')
    solver.config.options['hessian_approximation'] = 'limited-memory'
    results = solver.solve(m, tee=show_solver_log)
    pyo.assert_optimal_termination(results)
```

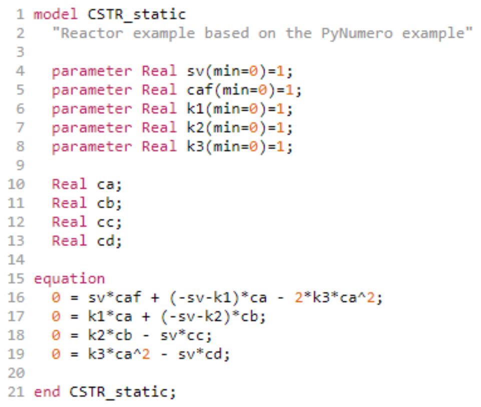
Figure 4. PyNumero reactor model optimization problem statement in Pyomo code

Notice that an instance of a Pyomo *ConcreteModel* class, `m`, is passed to an instance of the *SolverFactory* class, solver. This will be the same when using the new FMU/Pyomo interface package. Also notice that the values of `k1`, `k2`, `k3`, and `caf` are handled as equality constraints. Only `sv` is free to iterate during optimization. Pyomo hands this problem to cyipopt to solve the problem and produces the following print out of the results:

```
inputs : Size=5, Index=reactor._input_names_set
    Key : Lower : Value                  : Upper : Fixed : Stale : Domain
    caf :  None :                10000.0 :  None : False : False :  Reals
     k1 :  None :     0.8333333333333334 :  None : False : False :  Reals
     k2 :  None :     1.6666666666666667 :  None : False : False :  Reals
     k3 :  None : 0.00016666666666666666 :  None : False : False :  Reals
     sv :     0 :     1.3438109305149577 :  None : False : False :  Reals
outputs : Size=4, Index=reactor._output_names_set
    Key : Lower : Value              : Upper : Fixed : Stale : Domain
     ca :     0 :   3874.25779977848 :  None : False : False :  Reals
     cb :     0 :   1072.43720049758 :  None : False : False :  Reals
     cc :     0 : 1330.0943532862536 :  None : False : False :  Reals
     cd :     0 : 1861.6053232188908 :  None : False : False :  Reals
```

Figure 5. PyNumero reactor model solution results log

This shows that the solver converged to a solution of 1.34 with 1072 as the maximum concentration of species B. Alternatively, the problem can be written in another language, from which an FMU can be generated. For example, in Modelon Impact the reactor model definition could be written in Modelica as:



```
model CSTR_static
  "Reactor example based on the PyNumero example"

  parameter Real sv(min=0)=1;
  parameter Real caf(min=0)=1;
  parameter Real k1(min=0)=1;
  parameter Real k2(min=0)=1;
  parameter Real k3(min=0)=1;

  Real ca;
  Real cb;
  Real cc;
  Real cd;

equation
  0 = sv*caf + (-sv-k1)*ca - 2*k3*ca^2;
  0 = k1*ca + (-sv-k2)*cb;
  0 = k2*cb - sv*cc;
  0 = k3*ca^2 - sv*cd;

end CSTR_static;
```

Figure 6. Static continuously stirred reactor model in Modelica

Notice that the variables include attributes like `min`, `max`, `start`, `nominal`. These can be included in the FMU's variable attributes and the new FMU/Pyomo interface package will use these to automatically define the required Pyomo attributes. The interface also allows the attributes to be provided in the problem statement for cases when the FMU does not include these.

It is also worth noting that an added benefit to encoding this model in Modelica is that the Modelica translator can "tear" (Baharev et al., 2016) the system of equations to minimize the size of equation blocks. In this case the block is torn to a size 1 system around variable `ca`.

With this model the solution space of can be explored by plotting the solution of `cb` as a sweep of values of `sv`. Notice the location and maximum of this curve correlates to the solution from Ipopt shown in Figure 5.

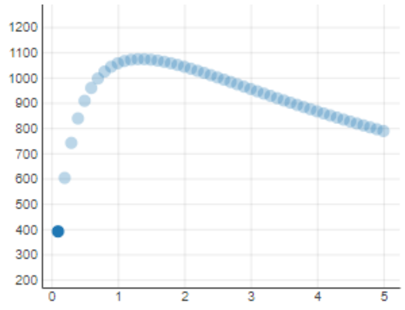
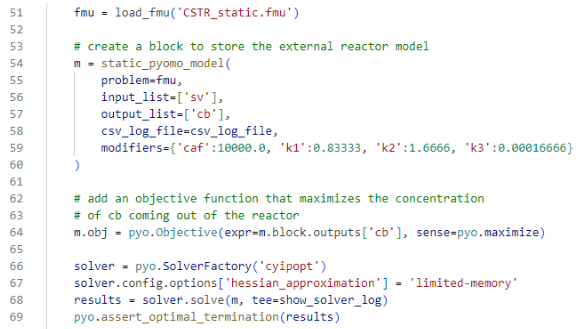


Figure 7. Sweep of outlet concentration, `cb`, (vertical axis) versus inlet flow rate, `sv` (horizontal axis)

A similar optimization problem can now be defined using the new FMU/Pyomo interface package. The package includes helper functions `static_pyomo_model` and `generic_pyomo_model` that return an instance of the PyNumero *ExternalGreyBoxBlock*, "`block`", as an element of an instance of a Pyomo *ConcreteModel*. The block contains an `external_model` that is an instance of one of either of the new FMU/Pyomo classes, *StaticFMU* or *GenericFMU*, as described in section 3. The methods in these classes are optimized for the type of FMU that is being used. An example of this is shown in the following code where `m` is the instance of a Pyomo *ConcreteModel* class that includes an instance of the *StaticFMU* class.



```
51      fmu = load_fmu('CSTR_static.fmu')
52
53      # create a block to store the external reactor model
54      m = static_pyomo_model(
55          problem=fmu,
56          input_list=['sv'],
57          output_list=['cb'],
58          csv_log_file=csv_log_file,
59          modifiers={'caf':10000.0, 'k1':0.83333, 'k2':1.6666, 'k3':0.00016666}
60      )
61
62      # add an objective function that maximizes the concentration
63      # of cb coming out of the reactor
64      m.obj = pyo.Objective(expr=m.block.outputs['cb'], sense=pyo.maximize)
65
66      solver = pyo.SolverFactory('cyipopt')
67      solver.config.options['hessian_approximation'] = 'limited-memory'
68      results = solver.solve(m, tee=show_solver_log)
69      pyo.assert_optimal_termination(results)
```

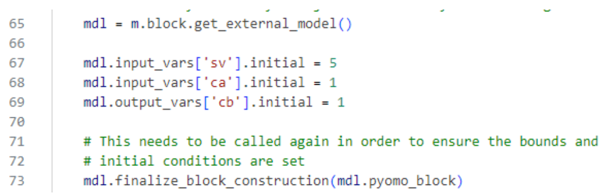
Figure 8. Optimization problem statement using the CSTR_static FMU.

By default the FMU's inputs and outputs will be used for the problem inputs and outputs, or they can be specified directly for FMUs without explicit I/O variables, as in the above code that specified `sv` and `cb` as the inputs and outputs respectively. Notice that the code that defines the optimization problem, i.e. the objective, solver, and its settings on lines 64 – 69 in the above code, is the same as lines 38 – 43 shown in Figure 4. Calling the `pprint` method of the Pyomo *ConcreteModel* instance, `m` prints the following information to the screen. This demonstrates that the found solution matches the solution shown in Figure 5.

```
inputs : Size=2, Index=block._input_names_set
    Key : Lower    : Value              : Upper  : Fixed : Stale : Domain
     sv :      0.0 : 1.3437710786561108 : 1e+100 : False : False :  Reals
outputs : Size=1, Index=block._output_names_set
    Key : Lower    : Value              : Upper  : Fixed : Stale : Domain
     cb : -1e+100 : 1072.4690498456466 : 1e+100 : False : False :  Reals
```

Figure 9. Solution log for the CSTR system using the static FMU/Pyomo interface

At this point it is important to note a difference in solver statistics between the two cases. In the original PyNumero example, Ipopt reports 13 iterations required to solve the problem. With the FMU based approach the solution required 17 iterations. The exact cause for this difference has not yet been determined but some differences can be noted. One difference with the previous example is the initialization. The original PyNumero example initializes the problem with `sv` = 5 and `ca` and `cb` both equal to 1 whereas the above FMU based example initialized at `sv` = 1 and `ca` and `cb` equal to 0. This can be changed in the problem statement by adding the following lines.

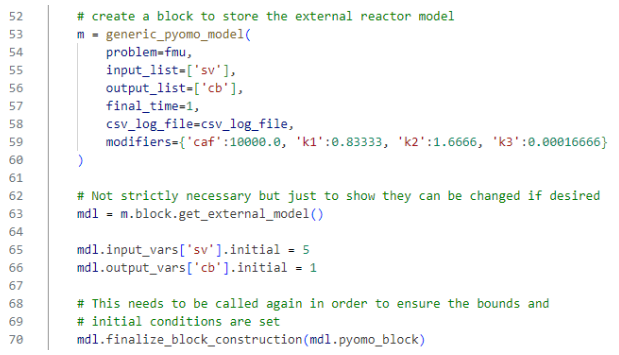


```
mdl = m.block.get_external_model()

mdl.input_vars['sv'].initial = 5
mdl.input_vars['ca'].initial = 1
mdl.output_vars['cb'].initial = 1

# This needs to be called again in order to ensure the bounds and
# initial conditions are set
mdl.finalize_block_construction(mdl.pyomo_block)
```

Figure 10. Applying consistent initial conditions to the problem statement

This reduces the number of iterations to 16 but still not to 13. This result is unexpected since the FMU based approach has both reduced the number of iteration variables and provides an analytic Jacobian. Resolution of this discrepancy would likely require low-level review of the verbose Ipopt log.

The above FMU based approach relied on the steady-state interface of the new FMU/Pyomo interface as described in the Introduction section. This allows the Ipopt solver to resolve the nonlinear block for the concentration of species A as described above. An alternative approach relies on the generic FMU/Pyomo interface described in the Introduction section. This approach supports any FMU, not just FMUs generated by Modelon Impact's steady-state interface. This is demonstrated with a generic FMU based on the model shown in Figure 6 and uses the *generic_pyomo_model* method of the FMU/Pyomo interface. Notice that the `ca` variable is no longer an input but is now an internal variable that will be resolved by the methods within the FMU.

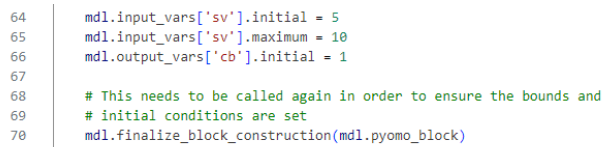


```
# create a block to store the external reactor model
m = generic_pyomo_model(
    problem=fmu,
    input_list=['sv'],
    output_list=['cb'],
    final_time=1,
    csv_log_file=csv_log_file,
    modifiers={'caf':10000.0, 'k1':0.83333, 'k2':1.6666, 'k3':0.00016666}
)

# Not strictly necessary but just to show they can be changed if desired
mdl = m.block.get_external_model()

mdl.input_vars['sv'].initial = 5
mdl.output_vars['cb'].initial = 1

# This needs to be called again in order to ensure the bounds and
# initial conditions are set
mdl.finalize_block_construction(mdl.pyomo_block)
```

Figure 11. Comparable problem statement for the CSTR system using the generic FMU/Pyomo interface.

Because generic FMUs can have time varying equations, the *simulate* method of the PyFMI FMU object is used to evaluate the variables. For model exchange FMUs this will attach an external DAE (differential algebraic equation) solver to integrate forward in time from zero to some user defined final time. For co-simulation FMUs, the DAE solver is internal so the simulate method requests the solver step forward from zero to the final time. The values returned to the optimization solver (e.g. Ipopt) are the values at the end of the simulation, even if steady-state conditions have not been reached. This also means that the Jacobian must be computed numerically.

Passing this problem to the solver results in a failed solution because the maximum number of iterations is reached. Adding a constraint on the upper bound of the iteration variable, `sv`, helps resolve this issue.

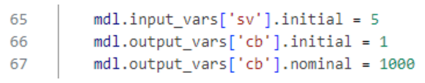


```
mdl.input_vars['sv'].initial = 5
mdl.input_vars['sv'].maximum = 10
mdl.output_vars['cb'].initial = 1

# This needs to be called again in order to ensure the bounds and
# initial conditions are set
mdl.finalize_block_construction(mdl.pyomo_block)
```

Figure 12. Defining the upper bound on the iteration variable to resolve failed solutions.

With an upper limit on the iteration variable, the solution now converges to the same solution as before but requires more than 2000 iterations.

Notice the iteration variable, `sv`, is on the order of 1.0 while the output concentration variable, `cb`, is on the order of 1000. This can significantly affect the accuracy of the Jacobian, especially when it is computed through the finite difference. Instead of applying a bound on the iteration variable, we can define better nominal values of the variables. The FMU/Pyomo interface will then scale the inputs, outputs, and Jacobian elements based on these nominal values.

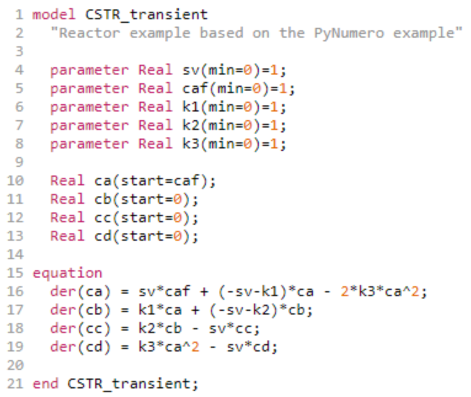


```
mdl.input_vars['sv'].initial = 5
mdl.output_vars['cb'].initial = 1
mdl.output_vars['cb'].nominal = 1000
```

Figure 13. Defining a nominal attribute for the output variable of the CSTR

In this case the solver converges to the solution much faster, although still not as fast as the case when an analytic Jacobian is used.

The last example using the CSTR model demonstrates the same problem but with a transient model. In this example the species concentrations are states of the differential equation system that evolve in time.



```
model CSTR_transient
  "Reactor example based on the PyNumero example"

  parameter Real sv(min=0)=1;
  parameter Real caf(min=0)=1;
  parameter Real k1(min=0)=1;
  parameter Real k2(min=0)=1;
  parameter Real k3(min=0)=1;

  Real ca(start=caf);
  Real cb(start=0);
  Real cc(start=0);
  Real cd(start=0);

equation
  der(ca) = sv*caf + (-sv-k1)*ca - 2*k3*ca^2;
  der(cb) = k1*ca + (-sv-k2)*cb;
  der(cc) = k2*cb - sv*cc;
  der(cd) = k3*ca^2 - sv*cd;

end CSTR_transient;
```

Figure 14. Modelica definition of the CSTR system that includes transient effects.

Simulating this for 1 second demonstrates these trajectories.

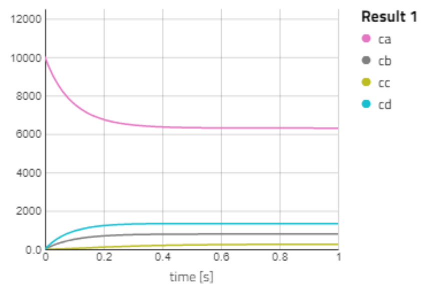
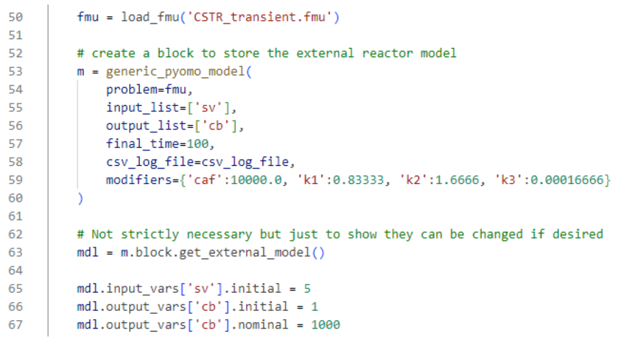


Figure 15. Species concentrations (vertical axis) transient response of the CSTR system versus time (horizontal axis)

Notice that the `generic_pyomo_model` function is used to define the problem statement. We also allow the simulation to proceed longer, to ensure that steady state conditions have been reached. Alternatively, we could have elaborated on the Modelica model to terminate the simulation after acceptable steady-state conditions have been reached. The problem statement remains the same as the other cases and the results are also comparable to the static cases.



```
fmu = load_fmu('CSTR_transient.fmu')

# create a block to store the external reactor model
m = generic_pyomo_model(
    problem=fmu,
    input_list=['sv'],
    output_list=['cb'],
    final_time=100,
    csv_log_file=csv_log_file,
    modifiers={'caf':10000.0, 'k1':0.83333, 'k2':1.6666, 'k3':0.00016666}
)

# Not strictly necessary but just to show they can be changed if desired
mdl = m.block.get_external_model()

mdl.input_vars['sv'].initial = 5
mdl.output_vars['cb'].initial = 1
mdl.output_vars['cb'].nominal = 1000
```

Figure 16. Transient CSTR optimization problem statement

Now that we have demonstrated the use of the FMU/Pyomo interface package with the CSTR model, we can use this for larger models. The results are shown for two different models. The first is a model from Modelon’s Jet Propulsion Library that simulates a steady-state operating gas turbine engine. The sizing of the engine is designed to meet the requirements for an example aircraft under the conditions of a rolling-take-off (RTO), top-of-climb (TOC), and cruise (CRZ) conditions. These requirements are constraints within the model and simultaneous evaluation of these three cases relies on three instances of the engine component within a single simulation model. This is based on the multi-point methodology defined by Kyprianidis et al. (2014). The optimization problem is to minimize the specific fuel consumption under cruising conditions.

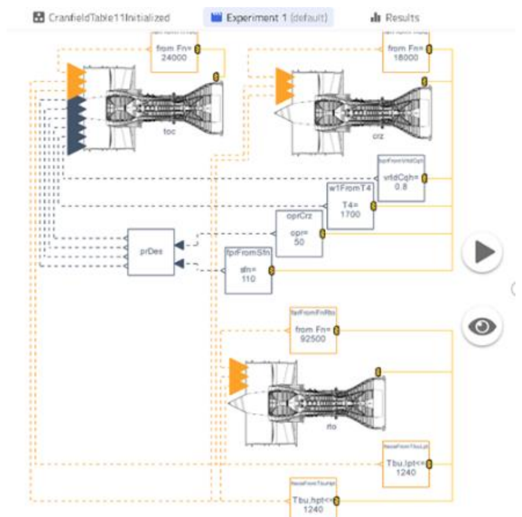


Figure 1. The Steady-state gas-turbine design model used for testing.

Sweeping some of the variables of interest and plotting the specific fuel consumption for the cruising case shows the local minimum (the solution that the solver will search for) occurs between 0.94 and 0.96 of the swept variable.

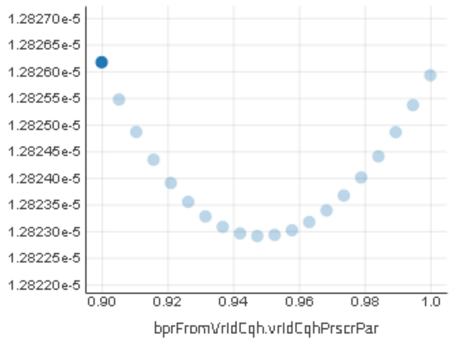
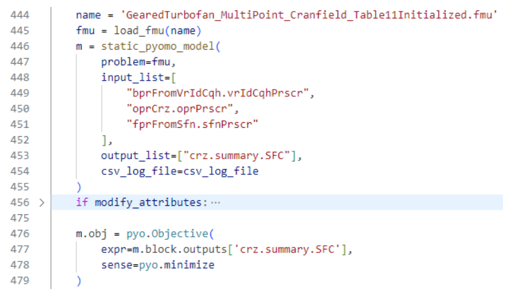


Figure 17. Specific fuel consumption (vertical axis) versus swept by-pass ratio constraint on the velocity ratio tuning variable (horizontal axis)

Defining the optimization problem is the same as for the CSTR system but with specifics for the FMU and its variables. The variables to tune are the cold-to-hot air velocity ratio, the overall pressure ratio of the engine, and the specific thrust. The nominal values for these (used for the normalization) are 1, 50, and 100 respectively.



```python
name = 'GearedTurbofan_MultiPoint_Cranfield_Table11Initialized.fmu'
fmu = load_fmu(name)
m = static_pyomo_model(
    problem=fmu,
    input_list=[
        "bprFromVrIdCqh.vrIdCqhPrscr",
        "oprCrz.oprPrscr",
        "fprFromSfn.sfnPrscr"
    ],
    output_list=["crz.summary.SFC"],
    csv_log_file=csv_log_file
)
if modify_attributes:···

m.obj = pyo.Objective(
    expr=m.block.outputs['crz.summary.SFC'],
    sense=pyo.minimize
)
```

Figure 18. Gas turbine specific fuel consumption minimization problem statement

The total problem involves 112 variables. Most of the iteration variables are the constraints needed to solve the design problem that requires the engine to meet the specific performance requirements. The solver converges to the solution within about 60 iterations.

The last model described in this paper defines an open-cycle parallel double-effect absorption heat pump (DEAHP) coupled to a multi-effect distillation (MED) system for brine desalination. In this system, the DEAHP acts as a steam recompression unit that improves the coupled system's overall efficiency proportionally to the coefficient of performance of heating (COPh).

The optimization problem is to maximize the COPh of the DEAHP by varying the strong and weak mass fraction concentrations of the absorption fluid `Xstrong` and `Xweak`, respectively.

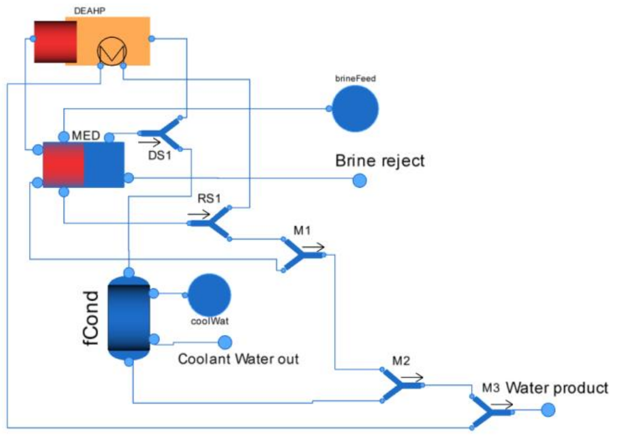


Figure 2. The Steady-state design model for a double effect absorption heat pump from (Stuber et al., 2015)

Using the new FMU/Pyomo interface package to define an optimization problem to solve with Ipopt results in convergence to a solution of 0.43 and 0.25 for `Xstrong` and `Xweak`, respectively with a COPh of 2.66. In this test, the starting values for `Xstrong` and `Xweak` were 0.588 and 0.549, respectively.

Sweeping the concentrations in simulation and observing the COPh can be used to verify the solution. Notice there is a local maximum in `Xstrong` and a maximum at the lower boundary of `Xweak`. This is consistent with the solution found by Ipopt.

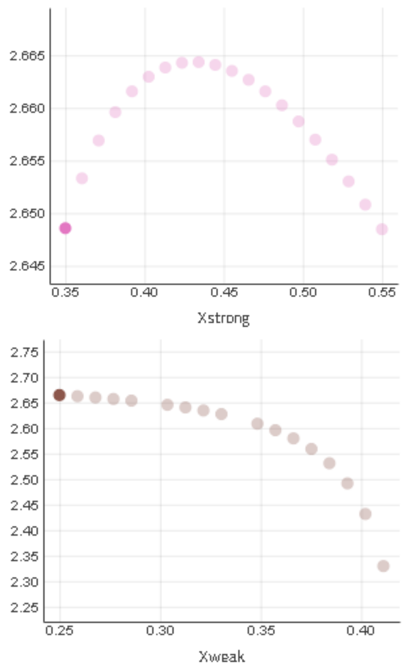


Figure 19. Example sweep of the strong and weak concentrations (horizontal axes) and the resulting coefficient of performance (vertical axes) that demonstrates the optimization problem's solution.

Table 1 lists some statistics related to the different models described in this paper. The key for the rows is shown in Table 2.

Table 1. Statistics per model.

CSTR Python	*CSTR FMU*	*Gas Turbine*	*DEAHP*
4	4	2382	621
4	1	109	30
1	1	3	2
13	16	60	24

Table 2. Row key for Table 1

1	Model implementation
2	Variable count
3	Equality constraint count
4	Optimization (tuner) variable count
5	Optimization iteration count

6 Conclusions

The Pyomo interface package has been used to solve both small and large optimization problems. There are two additional findings from this work. The first is the sensitivity of success to converge to a solution, on the

Jacobian. As described in the **Test Models and Results** section, small changes in the method used to estimate or compute the Jacobian had a significant effect on the ability of the solver to find a solution. This was demonstrated with a small example problem but is especially true as the models become larger.

Another significant finding was the importance of normalizing the variables. Even small models can benefit from this. Using the nominal attributes to normalize variables balances the sensitivity of individual variables so they do not dominate the convergence criteria. The current implementation uses the nominal attributes of the FMU variables automatically to normalize both the outputs and the Jacobian (using nominal attributes of both inputs and outputs).

Finally, models that can produce failed simulations or solutions can reduce the robustness of the solver. The current implementation returns a Numpy `NaN` value to the optimization solver in these cases. A possible improvement to this is proposed in the next section.

7 Future Work

Future developments for this work include:

- Native support for parameter tuning (including calibration) with respect to experimental data
- Support for dynamic optimization
- Sensitivity analysis to help the user identify tuner variables
- Improved support for failed solutions
- Testing and support for additional problem types and solvers (in addition to Ipopt that was used for this current work)

In addition to the above list of improvements to the Pyomo Interface Package, there is one item related to FMU creation. The Pyomo Interface Package natively supports FMUs generated with Modelon's steady-state interface for solving NLP problems. A proposed extension, similar to the steady-state interface, would be automatic conversion of mathematical operators with restricted domains into a more optimization friendly format. As described in Wächter 2009, mathematical functions with restricted domains can usually be converted into inequality constraints. It should be possible for Modelica translators to make this conversion automatically. This would make it easy for model developers to write the equations in a familiar format for normal simulation but export the FMU targeted for optimization when needed, similar to exporting an FMU for steady state evaluation.

Disclaimer

The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Acknowledgements

This material is based upon work supported by the National Alliance for Water Innovation (NAWI), funded by the U.S. Department of Energy, Office of Energy Efficiency and Renewable Energy (EERE), Industrial Efficiency and Decarbonization Office, under Funding Opportunity Announcement DE-FOA-0001905.

A special thanks to Professor George Bollas from the University of Connecticut for also supporting this work.

References

Åkesson, Johan and K-E Årzén, Magnus Gäfvert, Tove Bergdahl, Hubertus Tummescheit (2011). "Modeling and optimization with Optimica and JModelica. org—Languages and tools for solving large-scale dynamic optimization problems", Computers & Chemical Engineering, Volume 34, Issue 11, pp. 1747-1849.

Andersson, Joel and Johan Åkesson, Francesco Casella, Moritz Diehl (2011). "Integration of CasADi and JModelica.org". In: Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical University; Dresden; Germany, pp. 218-231. DOI: 10.3384/ecp11063218.

Baharev, Ali and Hermann Schichl, Arnold Neumaier (2016). "Tearing systems of nonlinear equations I. A survey." URL: https://api.semanticscholar.org/CorpusID:51987111

Bachmann, Bernhard and Lennart Ochel, Vitalij Ruge, Mahder Gebremedhin, Peter Fritzson, Vaheed Nezhadali, Lars Eriksson, and Martin Sivertsson (2012). "Parallel multiple-shooting and collocation Optimization with OpenModelica". In: Proceedings of the 9th International Modelica Conference. Linköping University Electronic Press, September 2012, pp. 659-668. DOI:10.3384/ecp12076659.

Bryson, Arthur E. Jr. (1999), Dynamic Optimization, Addison Wesley Longman, Inc. ISBN: 0-201-59790-X.

Bynum, Michael L. and Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, David L. Woodruff (2021). "Pyomo – Optimization Modeling in Python" 3rd Edition, 2021, ISSN 1931-6828.

Fletcher, R (1987). "Practical Methods of Optimization", 2nd ed. John Wiley and Sons. ISBN: 0-471-49463-1.

Kyprianidis, Konstantinos G. and Andrew M. Rolt, Tomas Grönstedt (2014). "Multi-Disciplinary Analysis of a Geared Fan Intercooled Core Aero-Engine", Journal of Engineering for Gas Turbines and Power, January 2014

Magnusson, Fredrick and Johan Åkesson (2015). "Dynamic Optimization in JModelica.org". In: Processes 2015, 3, pp. 471-496; DOI:10.3390/pr3020471.

Modelica Association (2024), FMI website, https://fmi-standard.org.

Modelon Impact Help Center (2024), website URL: https://help.modelon.com/latest/reference/oct/#dynamic-optimization-of-daes-using-direct-collocation-with-casadi.

OpenModelica (2024) website URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/optimization.html.

Rodriguez, Jose Santiago and Michael Bynum, Carl Laird, Bethany Nicholson, Robby Parker, John Siirola (2024). “PyNumero is a package for developing parallel algorithms for nonlinear programs”, https://pyomo.readthedocs.io/en/stable/contributed_packages/pynumero/index.html

Stuber, Matthew D. and Christopher Sullivan, Spencer A. Kirk, Jennifer A. Farrand, Philip V. Schillaci, Brian D. Fojtasek, Aaron H. Mandell (2015). “Pilot demonstration of concentrated solar-powered desalination of subsurface agricultural drainage water and other brackish groundwater sources” Desalination, Volume 355, 2015, Pages 186-196, https://doi.org/10.1016/j.desal.2014.10.037.

Ruge, Vitalij and Willi Braun, Bernhard Bachmann, Andrea Walther, and Kshitij Kulshreshtha (2014). "Efficient implementation of collocation methods for optimization using OpenModelica and adol-c". In: Proceedings of the 10th International Modelica Conference. Modelica Association and Linköping University Electronic Press, March 2014, pp. 1017-1025. DOI:10.3384/ecp140961017.

Wächter, Andreas and L. T. Biegler (2006). “On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming”, Mathematical Programming, 106(1), pp. 25-57, preprint at http://www.optimization-online.org/DB_HTML/2004/03/836.html

Wächter, Andreas (2009). “Short Tutorial: Getting Started With Ipopt in 90 Minutes”, In Combinatorial Scientific Computing. Dagstuhl Seminar Proceedings, Volume 9061, pp. 1-17, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2009), https://doi.org/10.4230/DagSemProc.09061.16